第4章 SystemVerilogで再利用性の高い検証環境を構築

PCIバスを例に，バス・レベル検証とシステム・レベル検証のサンプル記述を公開

宮下晴信

ここでは，SystemVerilogを使った検証環境の構築法について解説する．PCIバス・システムの検証を例に，サンプル記述を示しながら，バス・レベル検証とシステム・レベル検証のポイントを説明する．SystemVerilogは，従来のVerilog HDLと比べて検証記述の言語仕様が大幅に強化された．SystemVerilogの導入や活用を成功させるためには，再利用性の高い検証環境を構築することが不可欠といえる．（編集部）

今年（2005年）は，IEEE（The Institute of Electrical and Electronics Engineers, Inc.）で二つの言語SystemCとSystemVerilogの標準化が行われています．数年前は，「SystemC vs. SystemVerilog」という構図で論じられることもありましたが，昨年（2004年）あたりから，SystemCはTLM（transaction level modeling）をベースとしたアーキテクチャ設計やモデリングに，SystemVerilogはRTL（register transfer level）をベースとした実装設計（インプリメンテーション）および機能検証に使い分けるという流れになってきました（右掲のコラム「SystemVerilogのここが良い，ここが悪い」を参照）．

SystemVerilogの検証機能は，従来の検証言語（e言語やOpenVeraなど）と同等であり，言語仕様の中でアサーション（SystemVerilogアサーション）もサポートしています．また，C言語とのインターフェースについては，従来のVerilog HDLが持っていたPLI（programming language interface）/VPI（verilog programming interface）に加えて，DPI（direct programming interface）が用意されており，容易にC言語を利用できるようになりました．このようにSystemVerilogは，「検証を効率化するために作られた言語」と言っても過言ではありません．

SystemVerilogを適用すればアサーションやそのほかの

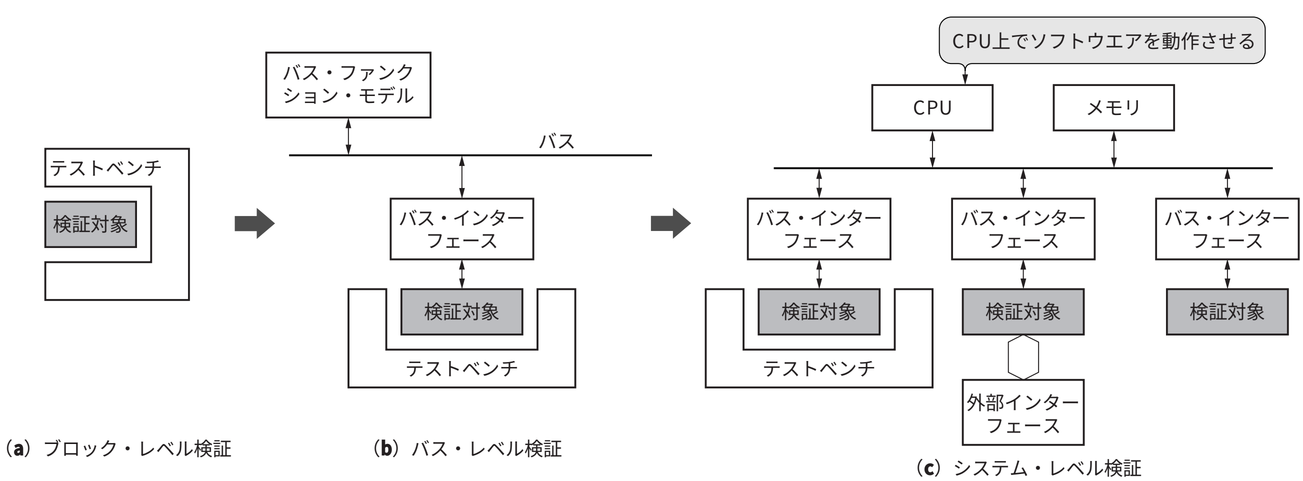


図1　機能ブロックの検証ステップ

機能ブロックの検証は，ブロック・レベル検証，バス・レベル検証，システム・レベル検証の三つのステップで行う．ブロック・レベル検証では機能ブロックを単体で検証する．バス・レベル検証では，機能ブロック＋バス接続回路をバス動作のレベルで検証する．システム・レベル検証は，CPUを含むシステムに機能ブロックを組み込んで検証する．

Column SystemVerilogのここが良い，ここが悪い

ここでは筆者が考えるSystemVerilogの良い点と不足している点について考えてみたいと思います．

良い点は以下の項目です．

- RTL記述を効率化できる．記述量が低減する．
- C/C++と同等のデータ・タイプとクラスを使える．
- 高度なモデリングに必要な機能（同期，イベント）がある．
- インターフェースが使える．
- 検証言語と同等の機能を備えている．
- アサーションが使える．
- 制約付きランダム・テスト生成が使える．
- 機能カバレッジが使える．
- C言語とのインターフェースを備えている．

従来までは，別途，検証言語を使わないとできなかったことが，SystemVerilogだけでできるようになったことがなんといっても良い点でしょう．

次に，不足していると筆者が考えている項目です．

- サポートするツールがまだ少ない．
- サポートする機能がツールによって異なる．
- 枯れていないので，ツールにバグが多い．
- 言語面でのTLM（transaction level modeling）のサポートが足りない．
- 方法論やスタイル・ガイド（例えばテストベンチ，アサーション，クラス，制約付きランダム・テスト生成，機能カバレッジの書きかたや使いかたなど）が確立されていない．
- 教科書が少ない（SystemVerilogに限らないが，とくに検証に関する文献が少ない）．

SystemVerilogの進化にツールはまだ追いついていません．今年（2005年）から各EDAベンダは本格的にSystemVerilogをサポートしてくると思いますので，期待しましょう．

SystemVerilogの検証言語と同等な機能は，従来の検証言語で行っていたものと考えかたが同じです．そこで，SystemVerilogに特化したスタイル・ガイドや教科書が出てくるまでは，従来の検証言語の使いかたをベースに運用していけばよいと思います．

C++の知識がないとなかなか使いこなせないSystemCなどと異なり，Verilog HDLや検証言語の延長として取り扱えるSystemVerilogは，LSI設計者にとって比較的取り組みやすい言語だと思います．あとは，ツールや方法論などの環境が整い，良いスタイル・ガイドが出てくれば，一気に広がっていくと思います．

機能が使えるといっても，それを使うための土台，つまり検証環境がなければ話は始まりません．そこで，本稿ではSystemVerilogを使って検証するための土台となる環境（検証環境）の構築について説明します．とくに，検証環境の構成要素であるトップ・テストベンチ，バス・ファンクション・モデル，テスト・プログラムを中心に解説します．

● 検証ステップは3段階

LSI設計における実装設計では，RTLのHDL記述を使って，あるまとまった機能を記述します．このあるまとまった機能のことを機能ブロックと呼びます．なお，機能ブロックは自社で開発したものだけでなく，IP（intellectual property）コアとして外部から調達することもあります．自社で開発した機能ブロックは，実装設計の後，検証しなければなりません．機能ブロックの検証は，次の三つのステップで行います（**図1**）．

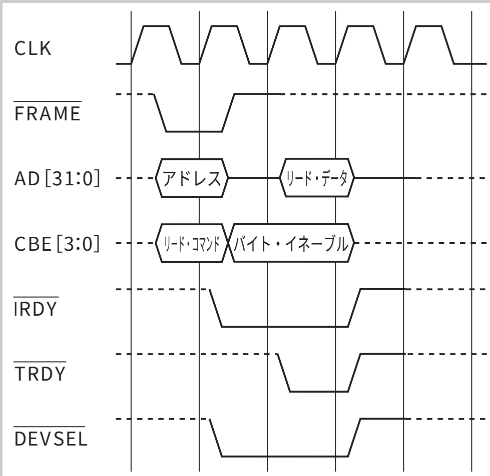
- **ブロック・レベル検証**
- **バス・レベル検証**
- **システム・レベル検証**

最初のステップは，機能ブロックを単体で検証するブロック・レベル検証です．ブロック・レベル検証では機能ブロック内のすべての機能を確認するだけでなく，機能ブロックと外部をつなぐインターフェース部のタイミングについても確かめます．つまり，ブロックの設計データだけで実施できる検証はすべて行います．

機能ブロックは，内部レジスタへのアクセスのためになんらかのバスに接続されます．バスに直接接続できない場合は，バス接続回路（ブリッジ回路，ラッパ回路とも呼ぶ）を使って間接的にバスに接続されます．この段階における検証が2番目のバス・レベル検証です．バス・レベル検証では，バス・ファンクション・モデルを使い，ブリッジ回路を経由して機能ブロックを検証します．なお，ブリッジ回路そのものについても，検証対象となっている機能ブロックと同じようにブロック・レベル検証を行う必要があります．バス・レベル検証では，ブリッジ回路と機能ブロックの間の通信プロトコルやタイミングの確認にバス・ファンクション・モデルを使用します．

最後のステップでは，バス・レベル検証の環境にCPUを追加し，CPUから機能ブロックにアクセスして正しく動作するかどうかを確認します．このステップを，ここではシステム・レベル検証と呼ぶことにします．システム・レベル検証は，おもにソフトウェアを使って行います．つまり，ハードウェア（機能ブロック）だけでなく，ソフトウェア（機能ブロックのファームウェアやデバイス・ドライバな

Column　PCIバスについておさらい



図A-1　PCIバスのプロトコル例（リード・コマンド）

PCIバスは33MHzで動作するパソコン用の32ビット汎用I/Oバスとして開発され，現在ではパソコンのみならず，組み込みシステムのI/Oバスとしても使われています．PCIバスはPCI SIG（PCI Special Interest Group，http://www.pcisig.com/）にて標準化が行われ，その後，66MHz，64ビットへの拡張，またサーバ用として性能を向上させたPCI-Xの策定，そして昨年，次世代PCI規格としてパソコン用チップセットに実装されたPCI Expressへと進化してきました．

PCIバスの最新のバージョンは，今年（2005年）リリースされたリビジョン3.0です．手元にある約10年前のInterface増刊 OPEN DESIGN No.7「PCIバスの詳細と応用へのステップ」（CQ出版社，1995年発行）によると，当時のPCIバスのリビジョンは2.0でした．つまり，約10年間，リビジョン2.xが使われてきたことになります．

コンピュータのバス・システムではマスタおよびスレーブという用語を使っていましたが，PCIバスではマスタのことをイニシエータ，スレーブのことをターゲットと呼びます．

図A-1にPCIバスのプロトコルの例（リード・サイクル）を示します．イニシエータがFRAME信号をアサートすることでサイクルが開

ど）の事前検証にもなります．とくにCPUを内蔵するシステムLSIの開発では，このシステム・レベル検証が重要になります．

本稿では三つの検証ステップのうち，2番目のバス・レベル検証の検証環境（トップ・テストベンチ，バス・ファンクション・モデル，テスト・プログラム）について，SystemVerilogによる作成方法を解説します．また，本誌2004年8月号，pp.80-90に掲載したSystemVerilogによる簡易

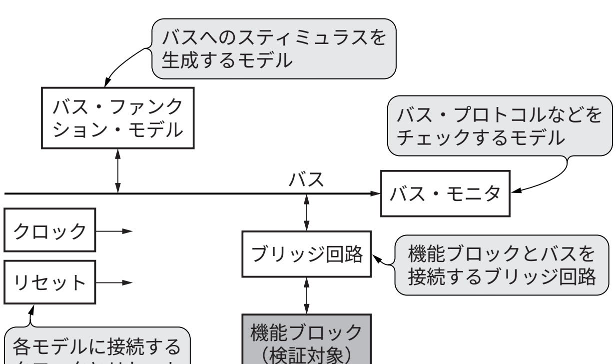


図2　バス・レベル検証のトップ・テストベンチ

検証対象となる機能ブロック，バスに与えるスティミュラスを生成するバス・ファンクション・モデル（BFM），バス・プロトコルなどをチェックするバス・モニタ，クロック部，およびリセット部から構成される．機能ブロックへのアクセスは，バス・ファンクション・モデルからブリッジ回路を経由して行われる．

CPUバス・モデルを使って，3番目のシステム・レベル検証を行うための検証環境について説明します．

● バス・ファンクション・モデルにinterfaceが使用可能

図2は，機能ブロックのバス・レベル検証におけるトップ・テストベンチ（最上位階層のテストベンチ）の例です．**図2**のトップ・テストベンチには，検証対象となる機能ブロック，バスに与えるスティミュラスを生成するバス・ファンクション・モデル（BFM），バス・プロトコルなどをチェックするバス・モニタ，クロック部，およびリセット部から構成されます．機能ブロックへのアクセスは，バス・ファンクション・モデルからブリッジ回路を経由して行われます．

バス・ファンクション・モデルからのアクセスにおいて，リード・アクセスではリード・データを期待値と比較し，ライト・アクセスではライト・データをテスト・データとして与えます．

従来，Verilog HDLではこのバス・ファンクション・モデルをモジュール（`module`）として実装し，モジュール内にバス・プロトコルを生成するタスクを記述していました．一方，SystemVerilogでは従来どおりモジュールを使ったバス・ファンクション・モデルのほかに，インターフェース（`interface`）を使ったバス・ファンクション・モデル

始されます．FRAME信号をアサートすると同時にAD信号にアドレスを，CBE信号にコマンドをドライブします．CBE信号は4ビットの信号で，ビットの割り当てによって**表A-1**に示すコマンドが使われます．イニシエータは次のデータ送信あるいは受信の準備ができるとIRDY信号をアサートします．一方，ターゲットはイニシエータからのアドレス信号およびコマンド信号をデコードし，自分のアドレスである場合はDEVSEL信号をアサートします．また，ターゲットはデータ送信あるいは受信の準備ができると，TRDY信号をアサートします．IRDY信号とTRDY信号がともにアサートされている場合にデータの転送が行われます．**図A-1**には明示していませんが，このほかにもいくつかの信号を使ってデータ転送を行います．

ここでは非常に簡単にPCIバスについて説明しました．詳細については仕様書で確認してください．参考資料として，CQ出版社発行のTECH I シリーズ Vol.21「改訂新版 PCIバス&PCI-Xバスの徹底研究」(2004年発行)がわかりやすいと思います．

表A-1　PCIバスのコマンド

	CBE[3]	CBE[2]	CBE[1]	CBE[0]
メモリ・リード	L	H	H	L
メモリ・リード・ライン	H	H	H	L
メモリ・リード・マルチプル	H	H	L	L
メモリ・ライト	L	H	H	H
メモリ・ライト・アンド・インバリデート	H	H	H	H
I/Oリード	L	L	H	L
I/Oライト	L	L	H	H
コンフィギュレーション・リード	H	L	H	L
コンフィギュレーション・ライト	H	L	H	H
インタラプト・アクノリッジ	L	L	L	L
スペシャル	L	L	L	H
デュアル・アドレス	H	H	L	H

を作成することができます．インターフェースは，SystemVerilogに導入された新しい機能です．インターフェースを利用すると，以下のことを期待できます．

- **コード記述量を大幅に削減できる**
- **設計の初期段階で接続などの詳細を決める必要がない**
- **モジュール間接続をカプセル化できる**

インターフェースでは，モジュールと同じようにタイム・スケールや信号宣言，assign文，タスク(task)，関数(function)，initial文などが使えます(**リスト1**)．

● PCIバス・モデルをバス・レベルで検証

バス・レベル検証の例題として，ここではPCI(Peripheral Component Interconnect)バスを取り上げます(上掲のコラム「PCIバスについておさらい」を参照)．なお，今回説明するバス・レベル検証のトップ・テストベンチやバス・ファンクション・モデルを含む検証環境は，ASICやシステムLSIの内部バス(オンチップ・バス)であるAMBA(Advanced Microcontroller Bus Architecture)やOCP(Open Core Protocol)などでも同じように使えます(バス・プロトコルや信号名が異なるだけ)．

図3に，検証対象であるデバイス(pci_dut)，pci_dutをテストするためのバス・ファンクション・モデル(pci_cpu)，インターフェース接続のブロック図を示します．また，**リスト2**に，トップ・テストベンチ(Testモジュール)のSystemVerilogコードを示します．pci_cpuモジュールのインスタンスcpuとpci_dutモジュールのインスタンスdutは，pci_interfaceインターフェースのインスタンスbusで接続されています．また，cpu，dut，busにはクロック部からクロック信号(CLK)，リセット部

リスト1　インターフェースの構文

```
interface <インターフェース名> ( <ポート宣言> );
    <timeunit/timepresion>;
    <信号，変数の宣言>;
    <assign文>;
    <initial文>;
    <task定義>;
    <function定義>;
endinterface [:<インターフェース名>]
```

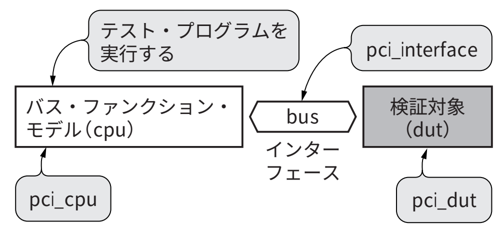


図3　PCIバスのトップ・テストベンチ

検証対象であるデバイス(pci_dut)，pci_dutをテストするためのバス・ファンクション・モデル(pci_cpu)，インターフェース接続から成る．

リスト2　トップ・テストベンチ

```
module Test;

  timeunit      1ns;
  timeprecision 100ps;

  logic         CLK, RST;

  Clock         clock( .CLK(CLK) );
  Reset         reset( .RST(RST) );

  pci_interface bus(   .CLK(CLK), .RST(RST) );

  pci_cpu       cpu(   .bus(bus), .CLK(CLK), .RST(RST) );
  pci_dut       dut(   .bus(bus), .CLK(CLK), .RST(RST) );

  final begin
    $display( “Test : finished at %0d ns”, $time );
  end

endmodule : Test
```

(a) Testモジュール

```
module Test;

  ....

  Clock         clock( .* );
  Reset         reset( .* );

  pci_interface bus(   .* );

  pci_cpu       cpu(    .* );
  pci_dut       dut(    .* );

  ....

endmodule : Test
```

(b) .*によるポート・バインド

リスト3　Verilog HDLによる記述

```
`timescale 1ns/100ps

module Test;

  wire         CLK, RST;
  tri1         FRAME;
  tri1         IRDY;
  tri1         TRDY;
  tri1         STOP;
  tri1         DEVSEL;
  tri1 [31:0]  AD;
  tri1 [3:0]   CBE;
  tri1         PAR;

  Clock        clock( .CLK(CLK) );
  Reset        reset( .RST(RST) );

  pci_cpu      cpu(   .CLK(CLK), .RST(RST),
                      .AD(AD), .CBE(CBE), .PAR(PAR),
                      .FRAME(FRAME), .IRDY(IRDY),
                      .TRDY(TRDY),
                      .STOP(STOP), .DEVSEL(DEVSEL) );

  pci_dut      dut(   .CLK(CLK), .RST(RST),
                      .AD(AD), .CBE(CBE), .PAR(PAR),
                      .FRAME(FRAME), .IRDY(IRDY),
                      .TRDY(TRDY),
                      .STOP(STOP), .DEVSEL(DEVSEL) );
endmodule
```

リスト4　インターフェースの内部信号をバインドする

```
pci_dut     dut(   .CLK(CLK), .RST(RST),

                   .AD(     bus.AD),     .CBE( bus.CBE),
                   .PAR(    bus.PAR),
                   .FRAME( bus.FRAME), .IRDY(bus.IRDY),
                   .TRDY(   bus.TRDY),   .STOP(bus.STOP),
                   .DEVSEL(bus.DEVSEL) );
```

からリセット信号(RST)が接続されています．

Testモジュール内でのタイム・スケールは，timeunitとtimeprecisionを使って1ns/100psにしています．この記述は，Verilog HDLの`timescaleディレクティブで1ns/100psを指定した場合と同じです．ただし，`timescaleディレクティブでは指定した場所から，次に`timescaleディレクティブで指定するまで同じタイム・スケールになりますが，timeunitとtimeprecisionは宣言したモジュールでのみ有効となります．finalブロックはSystemVerilogの新機能です．finalブロックは，$finishシステム・タスクが実行されることで呼び出されます．$finishシステム・タスクが出力するメッセージの後で，finalブロックが実行されます．**リスト2(a)**を実行した場合の結果では，シミュレーションの最後に終了したときの時間を表示します．

SystemVerilogでは，新しいポート・バインドを使うことにより記述をわかりやすくできます．**リスト2(b)**は，**リスト2(a)**に対して.*によるポート・バインドを使ったときの記述です．.*によるポート・バインドを使うことで，モジュール間の接続部の記述量が少なくなります．.*によるポート・バインドが行えるのは，モジュール側のポート名とそのポートに接続する信号名などが同じ場合に限られます．**リスト2(a)**では，すべてのポートが同じ信号名などになっているので，.*によるポート・バインドを利用できます．また，インスタンスcpuとdutのポート・バインドのように，一部のポートについて，従来どおりの名まえによるポート・バインドと組み合わせて使用することもできます．SystemVerilogで導入された新しいポート・バインドの詳細については，参考文献(1)，(2)，(4)を参照してください．

インターフェースを使わずに二つのモジュール間をPCI関連信号で接続すると，**リスト3**のようにcpuとdutを接続するためのPCI関連の信号の宣言やポートへの接続部分

リスト5　pci_interfaceインターフェース

```
interface pci_interface( input logic CLK, RST );

  timeunit      1ns;
  timeprecision 100ps;

  tri1        FRAME, IRDY, TRDY, STOP, DEVSEL, PAR;
  tri1 [31:0] AD;
  tri1 [3:0]  CBE;

  reg         FRAMEReg, IRDYReg, TRDYReg, STOPReg,
                                DEVSELReg, PARReg;
  reg  [31:0] ADReg;
  reg  [3:0]  CBEReg;

  assign FRAME  = FRAMEReg;
  assign IRDY   = IRDYReg;
  assign TRDY   = TRDYReg;
  assign STOP   = STOPReg;
  assign DEVSEL = DEVSELReg;
  assign PAR    = PARReg;
  assign AD     = ADReg;
  assign CBE    = CBEReg;

  task read_mem(  input bit [31:0] addr, input int size,
                 output bit [31:0] data );
      ....
  endtask : read_mem

  task write_mem( input bit [31:0] addr, input int size,
                  input bit [31:0] data );
      ....
  endtask : write_mem

  task read_io(   input bit [31:0] addr, input int size,
                 output bit [31:0] data );
      ....
  endtask : read_io

  task write_io(  input bit [31:0] addr, input int size,
                  input bit [31:0] data );
      ....
  endtask : write_io

  task read_cfg(  input bit [31:0] addr, input int size,
                 output bit [31:0] data );
      ....
  endtask : read_cfg

  task write_cfg( input bit [31:0] addr, input int size,
                  input bit [31:0] data );
      ....
  endtask : write_cfg

  initial begin
      FRAMEReg  = 1'bz;
      IRDYReg   = 1'bz
      TRDYReg   = 1'bz;
      STOPReg   = 1'bz;
      DEVSELReg = 1'bz;
      ADReg     = 32'bz;
      CBEReg    = 4'bz;
      PARReg    = 1'bz;
  end

endinterface : pci_interface
```

の記述が多くなり，非常にわかりにくくなります．前述したように，インターフェースを用いることでコード記述量を従来より大幅に削減できることがわかります．

実際の検証対象であるRTL記述は，**リスト3**のようにインターフェースを使用しない記述がほとんどでしょう．このような場合，**リスト4**のようにインスタンスdutのポートにインターフェース内の各信号を接続すれば，インスタンスcpuの部分は**リスト2(a)**，**(b)**と同じものがそのまま使えます．このような場合でも，部分的ではありますが，インターフェースの恩恵を受けられます．

● インターフェース内にバス・アクセス用タスクを実装

前述したようにVerilog HDLによるバス･ファンクション･モデルはモジュール内にバス・アクセス用のタスクを実装します．一方，SystemVerilogではインターフェース内にバス・アクセス用のタスクを実装します．

リスト5にPCIバスのインターフェース記述を示します．**リスト5**では，timeunitとtimeprecisionを使って，最初にタイム・スケールを1ns/100psと設定しています．これはPCIバスのクロック周期の30ns（周波数が33 MHz）を実現するためです．

次に，PCIバス関連信号の宣言です．ただし，ここではバス・リクエスト（BR），バス・グラント（BG），エラー（SERR/PERR），および割り込み（INTAなど）の信号は使っていません．なお，PCIバスは3ステートなので，各信号の型はtri1としています．また，ドライブする信号はreg型で宣言し，assign文を使って接続します．このように，マルチドライブを行う信号については，SystemVerilogでもVerilog HDLと同じ方法で記述する必要があります．

次の六つのタスク宣言は，PCIバスにおける三つのタイプのバス・サイクル（メモリ，I/O，コンフィグレーション）へのリードおよびライトを行うものです．ただし，**リスト5**ではタスクの詳細について記述していません．実際にこの検証環境を利用するときは，タスクの内容を定義してください．最後のinitial文は，reg宣言した信号を初期化しています．

● programを使ってテスト・プログラムを記述

テスト・プログラムは，バス・ファンクション・モデルのタスクを使って生成します．

テスト・プログラムは，従来のVerilog HDLのようにmoduleで記述することもできますが，ここではprogramを使用します．programを使うことにより，検証対象とテストベンチ（テスト・プログラム）の間のレース状態をなく

リスト6　テスト・プログラムの例

```
program pci_cpu( pci_interface bus,
                 input logic CLK,
                 input logic RST );

  bit [31:0] addr, data, exp;
  int        size;

  initial begin

    addr = 32'h0000_1000;
    size = 4;
    exp  = 32'h0011_2233;

    bus.write_mem( addr, size, exp  );
    bus.read_mem(  addr, size, data );

    if( data != exp )
      $display("compare error
     (exp:0x%08x) != data(0x%08x)",
                             exp, data);
  end

endprogram : pci_cpu
```

リスト7　modport

```
interface pci_interface( ( input logic CLK, RST );

  ....

  modport initiator( output FRAME, IRDY,
                     input  TRDY, STOP, DEVSEL,
                     inout  AD, CBE, PAR,

                     import task read_mem(  input  bit [31:0] addr,
                                            input  int        size,
                                            output bit [31:0] data ),

                     import task write_mem( input  bit [31:0] addr,
                                            input  int        size,
                                            input  bit [31:0] data ) );

                     import task read_io(   input  bit [31:0] addr,
                                            input  int        size,
                                            output bit [31:0] data ),

                     import task write_io(  input  bit [31:0] addr,
                                            input  int        size,
                                            input  bit [31:0] data ) );

                     import task read_cfg(  input  bit [31:0] addr,
                                            input  int        size,
                                            output bit [31:0] data ),

                     import task write_cfg( input  bit [31:0] addr,
                                            input  int        size,
                                            input  bit [31:0] data ) );

  modport target(    input FRAME, IRDY,
                     output  TRDY, STOP, DEVSEL,
                     ref  AD, CBE, PAR );

  ....

endinterface : pci_interface
```

▶(a) modport

すことができます．また，本稿では説明しませんが，クロッキング(clocking)といっしょに使うことにより，検証対象への信号の入出力タイミングを定義することもできます．

基本的に，programはmoduleと同じように記述できますが，大きな違いはalways文を使用できないということです．つまり，initial文でテスト・プログラムを書きます．また，module内の変数やタスクに外部から階層をまたいでアクセスできますが，program内の変数やタスクは外部からアクセスできません．アクセスするとエラーになります．

リスト6にPCIバスのテスト・プログラムの例を示します．moduleによる記述との違いは，module/endmoduleの代わりにprogram/endprogramを使うぐらいです．プログラム名のpci_cpuはmoduleにおけるモジュール名に相当します．この例では，initial文でPCIメモリ空間へライト・アクセスした後，リード・アクセスを行っています．PCIバスへのアクセスには，ポートbusに接続したpci_interface内のタスクを使います．ポートbus内のタスクにアクセスするには，bus.read_memのように .(ピリオド)を使います．

● modportによるポートの方向を指定する

インターフェースのインスタンスを各モジュールのポートにバインドするときですが，インターフェース内の信号の方向がわかりません．これを解決するための機能がmodportです．

リスト7(a)は，modport部分をpci_interfaceインターフェースに追加したものです．イニシエータのポートに接続するためのinitiatorとターゲットのポートに接続するためのtargetを定義しています．また，initiatorでは，接続したポート経由でモジュールがインターフェース内のタスクにアクセスできるようにするため，importを使って六つのタスクの宣言を追加しています．こうすることで，イニシエータとなるモジュールが，ポートにバインドしたpci_interfaceインターフェース内の六つのタスクにアクセスできます．initiatorに各タスクの宣言を追加せずにインターフェースのタスクにア

```
  module pci_cpu( pci_interface bus, input logic CLK, input logic RST );
      ....
  endmodule : pci_cpu

  module pci_dut( pci.interface bus, input logic CLK, input logic RST );
      ....
  endmodule : pci_dut


module Test;

  ....

  pci_cpu( .bus(bus.initiator), .* );
  pci_dut( .bus(bus.target),    .* );

  ....

endmodule : Test
```

◀(b) modportによるポート・バインド

▼(c) modportによるポートの型指定

```
module pci_cpu( pci_interface.initiator bus, input logic CLK,
                                                          input logic RST );
     ....
endmodule : pci_cpu

module pci_dut( pci.interface.target bus, input logic CLK,
                                                        input logic RST );
     ....
endmodule : pci_dut
```

クセスすると，エラーになります．

modportを使う場合，各インスタンスへのポート・バインドは，**リスト7(b)**のようにポートbusに接続する部分をインターフェースのインスタンス名(bus)からインターフェース内の各modport名に変更します．イニシエータ側では.bus(bus)を.bus(bus.initiator)に，ターゲット側では.bus(bus)を.bus(bus.target)に変更しています．

また，モジュール側のポートの型を変更することもできます．**リスト7(c)**では，ポートの型をpci_interfaceからpci_interface.initiator，およびpci_interface.targetに変更しています．この場合，各インスタンスのポート・バインド部は変更する必要はありません．

● 汎用インターフェースを利用する

リスト8のようにモジュールのポートに特定のインターフェース(pci_interface)を指定するのではなく，汎用インターフェース(generic interface)を指定することもできます．汎用インターフェースを使うことにより，バス・ファンクション・モデルを再利用できる可能性が広がります．

ただし，モジュール内でアクセスするタスクは，ポート(この例ではbus)に接続するインターフェース内で定義されている必要があるので注意してください．定義されていないタスクを使うとエラボレーションのときにエラーになります(ここでのエラボレーションとは，コンパイル後，シミュレーションが開始されるまでの間のことを指す)．

ポート・バインドは**リスト7(b)**と同じで，インターフェース内のmodport名を指定します．pci_cpuモジュールはイニシエータなのでinitiatorを，pci_dutモジュールはターゲットなのでtargetをポートにバインドします．

● CPUバス・モデルを再利用してシステム・レベル検証

ここまでで，バス・レベル検証におけるPCIバスの検証環境の各要素(トップ・テストベンチ，バス・ファンクション・モデル，テスト・プログラム)について説明しました．以降ではシステム・レベル検証について説明します．

本稿ではソフトウェアを使ったシステム・レベル検証を行います．システム・レベル検証ではソフトウェアを実行するCPUモデルが必要になります．CPUモデルは，本誌

リスト8
汎用インターフェースによるポートの型指定

```
// pci_interface.initiator busをinterface busに変更
module pci_cpu( interface bus, input logic CLK, input logic RST );
     ....
endmodule : pci_cpu

// pci_interface.target busをinterface busに変更
module pci_dut( interface bus, input logic CLK, input logic RST );
     ....
endmodule : pci_dut
```

4

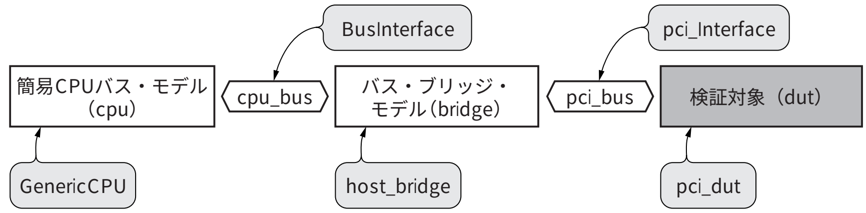


図4
簡易CPUバス・モデルとの接続

簡易CPUバス・モデルは，バス・ブリッジ・モデルを介して検証対象の`pci_dut`と接続する．簡易CPUバス・モデルとバス・ブリッジ・モデルの間は`BusInterface`インターフェースを，バス・ブリッジ・モデルと`pci_dut`の間は`pci_interface`インターフェースを使って接続する．

リスト9　簡易CPUバス・モデルを追加したトップ・テストベンチ

```
module Test_cpu;

  timeunit      1ns;
  timeprecision 10ps;

  logic         CPU_CLK, CPU_RST, PCI_CLK, PCI_RST;

  Clock  #(5)   sys_clock( .CLK(CPU_CLK) );
  Reset         sys_reset( .CLK(CPU_CLK), .RST(CPU_RST) );
  BusInterface  cpu_bus(   .CLK(CPU_CLK), .RST(CPU_RST) );

  GenericCPU    cpu(       .bus(cpu_bus.cpu),
                           .CLK(CPU_CLK), .nRST(CPU_RST) );

  Clock  #(15)  pci_clock( .CLK(PCI_CLK) );
  Reset         pci_reset( .CLK(PCI_CLK), .RST(PCI_RST) );
  pci_interface pci_bus(   .CLK(PCI_CLK), .RST(PCI_RST) );

  host_bridge   bridge(    .cpu_bus(cpu_bus.pci),
                           .CPU_CLK(CPU_CLK), .CPU_RST(CPU_RST),
                           .pci_bus(pci_bus.initiator),
                           .PCI_CLK(PCI_CLK), .PCI_RST(PCI_RST) );

  pci_dut       dut(       .bus(pci_bus.target),
                           .CLK(PCI_CLK), .RST(PCI_RST) );

  final begin
    $display( "Test : finished at %0d ns", $time );
  end

endmodule : Test_cpu
```

リスト10　BusInterfaceインターフェースの記述

```
interface BusInterface( input logic CLK,
                        input logic RST );

  modport cpu(
    import task Bus_Reset,
    import task Bus_Load
                ( input logic [31:0] ADDR,
                  output [31:0] DATA ),
    import task Bus_Store
                ( input logic [31:0] ADDR,
                  input  [31:0] DATA ) );

  modport pci(
    export task Bus_Load,
    export task Bus_Store );

  task Bus_Reset;
    $display("BusInterface : Bus_Reset");
  endtask : Bus_Reset

endinterface : BusInterface
```

2004年8月号，pp.80-90に掲載したSystemVerilogによる簡易CPUバス・モデルを利用します．これにより，簡易アセンブラ言語を使ってプログラムを作成できます．

図4のように簡易CPUバス・モデルは，バス・ブリッジ・モデルを介して検証対象の`pci_dut`と接続します．簡易CPUバス・モデルとバス・ブリッジ・モデルの間は`BusInterface`インターフェースを使って接続します．一方，バス・ブリッジ・モデルと`pci_dut`の間は`pci_interface`インターフェースを使って接続します．**図4**の構成におけるSystemVerilogの記述を**リスト9**に示します．

簡易CPUバス・モデルと`BusInterface`インターフェースは，PCIバスと違うクロック信号(`CPU_CLK`)およびリセット信号(`CPU_RST`)で動作します．クロック信号`CPU_CLK`については，`Clock`モジュールに対して，周期の半分が5nsになるようにパラメータとして`#(5)`を渡しています．

`BusInterface`インターフェースのポート名は本誌2004年8月号に掲載したものと同じですが，記述は**リスト10**のように変更しています．簡易CPUバス・モデルは`modport`を使って三つのタスク・ポートを`bus`経由で呼び出しています．今回の変更点は，このうちの二つのタスク(`Bus_Load`，`Bus_Store`)について，PCI側に接続するバス・ブリッジ・モデル(`host_bridge`モジュール)内のタスクを呼ぶようにしているところです．

このような方法は前述したインターフェース記述と多少異なるので，**図5**を使って詳しく説明します．簡易CPUバス・モデル内でポート`bus`経由で呼び出せるタスクは，`BusInterface`インターフェース内の`modport cpu`で`import`として宣言した`Bus_Reset`タスク，`Bus_Load`タスク，`Bus_Store`タスクです．`Bus_Reset`タスクを呼び出したときは，前述のコードと同じように`Bus_`

リスト11　バス・ブリッジ・モデル

```
module host_bridge( BusInterface cpu_bus,
                    input logic CPU_CLK, input logic CPU_RST,
                    pci_interface pci_bus,
                    input logic PCI_CLK, input logic PCI_RST );

  timeunit      1ns;
  timeprecision 10ps;

  task cpu_bus.Bus_Load(  input logic [31:0] ADDR, output [31:0] DATA );

    int n_addr, size;
    n_addr = ADDR & 32'hffff_f000;
    size = 4;

    #25;

    case(n_addr)
      32'h0000_0000 : pci_bus.read_mem(  n_addr, size, DATA );
      32'h0000_1000 : pci_bus.read_io(   n_addr, size, DATA );
      32'h0000_2000 : pci_bus.read_cfg(  n_addr, size, DATA );
      default : $display("Bus_Load  : illegal address (0x%08x)", ADDR );
    endcase;

    #25;

  endtask

  task cpu_bus.Bus_Store( input logic [31:0] ADDR, input [31:0] DATA );

    int n_addr, size;
    n_addr = ADDR & 32'hffff_f000;
    size = 4;

    #25;

    case(n_addr)
      32'h0000_0000 : pci_bus.write_mem( n_addr, size, DATA );
      32'h0000_1000 : pci_bus.write_io(  n_addr, size, DATA );
      32'h0000_2000 : pci_bus.write_cfg( n_addr, size, DATA );

      default : $display("Bus_Store : Illegal address (0x%08x)", ADDR );
    endcase

    #25;

  endtask

endmodule : host_bridge
```

(**a**) host_bridgeモジュールの記述

```
module host_bridge( ..... );

  .....

  parameter s_wait = 25, e_wait = 25;

  task cpu_bus.Bus_Load( .... );

     ....

     #s_wait;

     ....

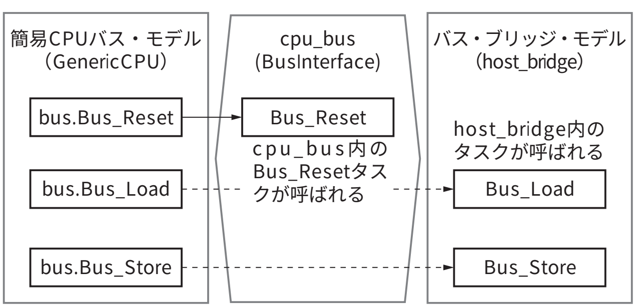
     #e_wait;

  endtask

  (cpu_bus.Bus_Storeタスクも同様になる)

endmodule : host_bridge
```

(**b**) ウェイトをパラメータ化した記述



▼図5
簡易CPUバス・モデルからのアクセス
簡易CPUバス・モデルはmodportを使って三つのタスク・ポートをbus経由で呼び出している．このうちの二つのタスク(Bus_Load，Bus_Store)について，PCI側に接続するバス・ブリッジ・モデル(host_bridgdeモジュール)内のタスクを呼ぶようにしている．

Interfaceインターフェース内のBus_Resetタスクが呼び出され，BusInterface:Bus_Resetが表示されます．一方，Bus_LoadタスクおよびBus_StoreタスクはBusInterfaceインターフェース内で定義されていません．その代わりにmodport pciでexportを使って宣言します．BusInterfaceインターフェース内のBus_LoadタスクあるいはBus_Storeタスクが呼び出された場合は，modport pciで接続されたモジュール内にあるBus_LoadタスクあるいはBus_Storeタスクが呼び出されることになります．つまり，cpu_bus.pciをポートに接続したhost_bridgeモジュールのbridgeインスタンス内にあるBus_LoadタスクあるいはBus_Storeタスクが呼ばれることになります．

リスト11(a)に，host_bridgeモジュールのコードを示します．Bus_Loadタスクについてはcpu_bus.Bus_Loadタスクを，Bus_Storeタスクはcpu_bus.Bus_Storeタスクを呼ぶように定義しています．こうすることでcpu_busポートに接続したcpu_busインスタンスのBus_LoadタスクおよびBus_Storeタスクを呼び出

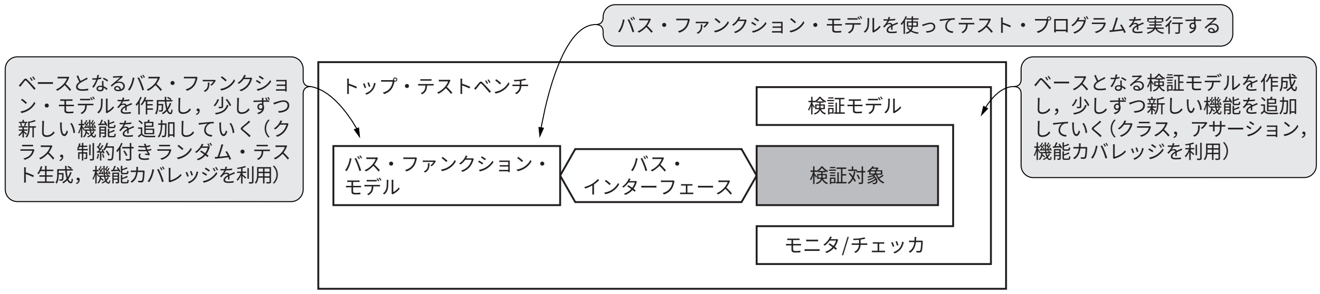


図6　検証環境

今回説明した検証環境をベースに，SystemVerilogのそのほかの検証機能（クラス，制約付きランダム・テスト生成，機能カバレッジ，アサーション）を組み込んでいけば，さらに使いやすいものになるだろう．

せるようになります．bus_bridgeモジュールの各タスクでは，アドレスの上位20ビットをデコードし，対応するアドレスに対してpci_busポートのタスク（Bus_Loadタスクではread_mem，read_io，read_cfg，Bus_Storeタスクではwrite_mem，write_io，write_cfg）を呼び出しています．対応するアドレスがない場合はエラー・メッセージを表示し，pci_busポートへのアクセスは行いません．

pci_busポートへのアクセスの前後に#25のウェイトを入れています．host_bridgeモジュール内でtimeunitを1nsとしているので，25nsのウェイトになります．こうすることで，簡易CPUバス・モデルからPCIバスへのアクセスに必要な時間は50ns（25ns＋25ns）＋PCIバスへのアクセス時間になります．また，**リスト11（b）**のように#25をパラメータ化し，#s_waitおよび#e_waitにすることにより，モジュールの外部からウェイト時間を設定することも可能となります．

以上，説明してきたように，簡易CPUバス・モデルを使ってPCIバスへのアクセスができるようになりました．あとは，簡易アセンブラを使って，PCIバスへのアクセス・プログラムを作成すればソフトウェアでテストすることができます．

● 検証環境も再利用を考えて構築

今回は，SystemVerilogの新しい機能であるインターフェースを使って，簡易的とはいえPCIバスに接続するデバイスをテストするための検証環境を構築しました．PCIバス部分をほかのバス（例えばAMBAやOCPなど）に変更することにより，基本的なモジュールの構成を変えることなく検証環境を再利用できます．

また，このような環境をベースに，SystemVerilogのそのほかの検証機能（クラス，制約付きランダム・テスト生成，機能カバレッジ，アサーション）をモデルに組み込んでいくことで，より使いやすい検証環境を構築できます（**図6**）．

実装設計のRTL記述は，IPコアとして再利用することがあたりまえになっていますが，検証用モデルやテストベンチなどの環境も再利用することを前提に作成していくべきだと思います．設計生産性だけでなく，検証を含めた開発全体の生産性を引き上げる必要があります．そのためには過去の資産をフルに再利用し，できる限りむだを省き，新規部分の設計や機能の追加に力を入れていくべきです．

参考・引用*文献

（1）SystemVerilog 3.1a Language Reference Manual，2004，http://www.eda.org/sv/SystemVerilog_3.1a.pdf.
（2）宮下晴信；SystemVerilogで簡易CPUバス・モデルを記述，pp.80-90，Design Wave Magazine，2004年8月号．
（3）宮下晴信；続・SystemVerilogで簡易CPUバス・モデルを記述，DPIを使ってC++モデルを接続する，pp.121-129，Design Wave Magazine，2004年12月号．
（4）Sutart Sutherland，Simon Davidmann，Peter Flake；System Verilog for Design，Springer（旧Kluwer Adademic Publisher），2004.

みやした・はるのぶ
富士ゼロックス（株）

<筆者プロフィール>
宮下晴信．今年もSystemVerilog，SystemCともに挑戦していきたいと思っている，自称「モデリング＆検証エンジニア」．